Programmation par objets et parallélisme de données dans Paladin

Frédéric Guidec

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Programmation par objets et parallélisme de données dans Paladin

Frédéric Guidec

N° 2393

Octobre 1994

PROGRAMME 1

ISSN 0249-6399

INRIA
RENNES

Programmation par objets et parallélisme de données dans Paladin

Frédéric Guidec*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Pampa

Rapport de recherche n° 2393 — Octobre 1994 — 19 pages

Résumé : Nous proposons d'exprimer la parallélisation par distribution des données et le modèle d'exécution SPMD à l'aide des mécanismes de la programmation par objets. Nous montrons qu'il est possible, en utilisant un langage à objets séquentiel, de développer des composants logiciels réutilisables décrivant à la fois la distribution des données et des algorithmes manipulant ces données de manière efficace. Pour illustrer notre propos, nous décrivons la bibliothèque PALADIN, conçue selon cette approche et dédiée au calcul d'algèbre linéaire sur machine parallèle.

Mots-clé : machines à mémoire distribuée, parallélisation par distribution des données, programmes SPMD, programmation par objets.

(Abstract: pto)

*. guidec@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Object–oriented programming and data–parallelism in Paladin

Abstract: We propose to express the data-parallelism and the SPMD execution model with the mechanisms of object-oriented programming. We show that it is possible to develop reusable software components that describe the distribution of data and algorithms that deal with these data efficiently, using a sequential object-oriented language. In order to illustrate this approach, we describe the PALADIN library, designed along this line and devoted to linear algebra computation on parallel machines.

Key-words: distributed memory machines, data-distribution, SPMD programs, object-oriented programming.

1 Introduction

Bien que la puissance de calcul des architectures parallèles à mémoire distribuée (APMD) atteigne aujourd'hui plusieurs dizaines de giga-flops, leur diffusion demeure encore très limitée en raison du manque de maturité des outils logiciels qui leur sont associés. Dans le projet EPEE (Environnement Parallèle d'Exécution de Eiffel [7, 8]), nous proposons d'exprimer le parallélisme de données et le modèle d'exécution SPMD à l'aide des mécanismes de la programmation par objets.

Le parallélisme de données, obtenu lorsque les données manipulées par un algorithme sont distribuées sur une APMD de manière à pouvoir être traitées en parallèle, est particulièrement bien adapté aux algorithmes manipulant des *agrégats*, c'est à dire de très grandes structures de données homogènes. Lorsque l'on combine la distribution d'agrégats avec le mode d'exécution SPMD (*Single Program Multiple Data*), le même code d'application peut être chargé sur tous les nœuds d'une APMD pour y être exécuté en parallèle, chaque nœud se contentant toutefois de ne traiter qu'une partie des données. On préserve ainsi la simplicité conceptuelle de la programmation séquentielle : le programmeur d'application se contente de spécifier la politique de distribution qu'il désire voir appliquer aux agrégats manipulés, mais ne gère pas explicitement le parallélisme afférant à cette distribution.

La programmation par objets [11] permet de développer des composants logiciels réutilisables décrivant, d'une part des agrégats pouvant être distribués sur une APMD, d'autre part des algorithmes manipulant ces agrégats de manière efficace. En utilisant les mécanismes de l'encapsulation et du masquage d'information, la distribution des agrégats et le parallélisme afférant peuvent être intégrés proprement dans des classes d'un langage à objets séquentiel.

Une première maquette de l'environnement EPEE, construite en 1991 sur la base du langage Eiffel [12], a permis de démontrer que l'on peut effectivement intégrer totalement la distribution des agrégats et le parallélisme SPMD dans les classes d'un langage à objets séquentiel [8]. EPEE est constitué, d'une part d'outils de compilation croisée permettant de générer du code exécutable pour diverses machines cibles, et d'autre part d'un ensemble de classes Eiffel qui fournissent au programmeur des mécanismes génériques facilitant la distribution des agrégats et leur manipulation en parallèle.

2 Développement d'une bibliothèque parallèle d'algèbre linéaire

2.1 Motivation

Afin de démontrer que l'approche que nous proposons dans le cadre du projet EPEE se prête parfaitement à la mise au point de bibliothèques de calcul pour machines parallèles, nous avons entrepris de développer une bibliothèque de démonstration baptisée PALADIN [6], permettant d'effectuer des calculs d'algèbre linéaire sur APMD. En développant cette bibliothèque, nous nous sommes efforcés de favoriser les critères suivants :

- **Extensibilité**
  Il doit à tout instant être possible d'enrichir la bibliothèque PALADIN de nouveaux algorithmes ou d'y adjoindre de nouveaux types de matrices et vecteurs, caractérisés par un nouveau format de représentation interne (*e.g.* matrices et vecteurs creux) ou par un nouveau schéma de distribution.

- **Transparence**
  Il s'agit avant tout d'assurer le confort du programmeur d'application, c'est à dire de l'utilisateur de la bibliothèque. À son niveau, les détails de mise en oeuvre – et notamment ceux relatifs à la gestion des matrices et vecteurs distribués – doivent être aussi transparents que possible. L'utilisateur se contente donc de spécifier, par exemple, les caractéristiques des matrices qu'il souhaite utiliser (denses ou creuses, distribuées ou non, etc.), et doit ensuite pouvoir manipuler toutes ces matrices de la même façon. Dans le cas des matrices distribuées, l'utilisateur devra ainsi spécifier le schéma de distribution désiré, mais n'aura pas à gérer explicitement cette distribution.

- **Performances et portabilité**
  Les performances de la bibliothèque doivent en faire une alternative intéressante aux autres approches actuelles, représentées notamment par les compilateurs-parallèliseurs semi-automatiques en cours de développement [1] et par les bibliothèques clés en main telles que ScaLapack [3]. Cependant la recherche de performances ne doit pas se faire au détriment de la portabilité. La bibliothèque devant pouvoir être aisément portée sur de nouvelles machines parallèles, elle doit être aussi indépendante que possible des caractéristiques de l'architecture sous-jacente.

2.2 Spécification abstraite des matrices et vecteurs

La bibliothèque PALADIN s'articule autour des classes MATRIX et VECTOR, qui renferment les spécifications abstraites des deux entités de base manipulées en algèbre linéaire[1].

Dans cet article nous nous concentrons sur la hiérarchie des classes décrivant les matrices, reproduite partiellement dans la figure 1. Dans la classe abstraite MATRIX, aucun détail n'est donné quant à la manière de représenter les matrices en mémoire. Ce genre de détail de mise en œuvre est volontairement laissé de côté, pour être fourni dans des classes descendantes. Dans la classe MATRIX on énumère simplement les attributs et méthodes permettant de manipuler un objet matrice, ainsi que leurs propriétés formelles exprimées sous la forme d'assertions (préconditions, postconditions, invariants, etc.).

```
deferred class MATRIX
feature -- Attributes
  nrow: INTEGER;      -- Number of rows
  ncolumn: INTEGER;   -- Number of columns
feature -- Accessors
  item (i, j: INTEGER): DOUBLE is
    -- Return current value of item(i, j)
  require
    valid_i: (i > 0) and (i <= nrow);
    valid_j: (j > 0) and (j <= ncolumn);
  deferred end;

  put (v: like item; i, j: INTEGER) is
    -- Put value v into item(i, j)
  require
    valid_i: (i > 0) and (i <= nrow);
    valid_j: (j > 0) and (j <= ncolumn);
  deferred
  ensure
    item (i, j) = v
  end;

feature -- Operators
  checksum: like item is
```

1. Toutes les classes constituant la bibliothèque PALADIN sont génériques. Cependant, afin de pas surcharger les figures et les extraits de code reproduits dans cet article, nous affectons de ne traiter que les matrices de nombres réels en double précision.

dire qu'ils sont déclarés, mais non définis [2]. On se contente d'indiquer leur signature (nombre et types des arguments) et leurs propriétés essentielles (préconditions et postconditions).

À la différence des accesseurs, les opérateurs tels que `checksum`, `add`, etc. sont des méthodes dont la définition peut être faite sur la base d'appels aux accesseurs, voire d'appels à d'autres opérateurs. La mise en œuvre d'un opérateur n'est donc pas directement dépendante du format de représentation choisi, celui-ci étant masqué par les accesseurs. Ainsi, dans la classe MATRIX, la fonction `checksum` calculant la somme de contrôle de la matrice courante est dotée de la définition suivante :

```
checksum: like item is   -- The value returned has
  local                  -- the same type as 'item'
    i, j: INTEGER;
  do
    from i := 1 until i > nrow loop
      from j := 1 until j > ncolumn loop
        Result := Result + item (i, j);
        j := j + 1;
      end; -- loop
      i := i + 1
    end; -- loop
  end;
```

La classe MATRIX contient ainsi un grand nombre d'opérateurs, permettant d'effectuer la plupart des opérations d'algèbre linéaire traditionnelles, telles que les opérations de type matrice-vecteur et matrice-matrice (somme, différence, produit, transposée, etc.), ainsi que les opérations plus complexes de factorisation LU, LDL^T, QR, la résolution de systèmes triangulaires, etc. Tous ces opérateurs sont mis en œuvre directement dans la classe MATRIX. Cette classe est donc qualifiée de classe abstraite, non pas parce qu'on n'y détaille aucune spécification opérationnelle, mais simplement parce qu'on y fait abstraction des détails de représentation en mémoire des objets manipulés.

Notons que la bibliothèque PALADIN est bien extensible, conformément à nos objetifs, dans la mesure où il demeure toujours possible d'ajouter dans la classe MATRIX un nouvel opérateur. Le mécanisme de l'héritage assure que cet ajout sera répercuté au niveau de toutes les classes descendantes. En outre, tout algorithme encapsulé dans la classe MATRIX sera perçu comme un algorithme par défaut, qu'il

2. Une méthode différée en Eiffel équivaut à une fonction virtuelle pure en C++.

sera toujours possible de redéfinir en fonction des besoins – par exemple dans un but de parallélisation des calculs – dans certaines classes descendantes.

2.3 Mise en œuvre des matrices locales

Les détails relatifs à la représentation en mémoire des matrices sont encapsulés dans des classes descendantes de MATRIX. La classe LOCAL_MATRIX décrit l'une des mises en œuvre possibles pour les matrices locales. Du point de vue d'un programmeur d'application, une matrice locale déclarée et créée au niveau d'un programme SPMD est une matrice locale à tous les nœuds, ce qui revient à dire qu'elle est dupliquée. Les instances de la classe LOCAL_MATRIX servent également de composants élémentaires pour la mise en œuvre des matrices distribuées, comme décrit dans le paragraphe 2.4.3

Une matrice de type LOCAL_MATRIX est représentée en mémoire sous la forme d'un simple tableau bi-dimensionnel. La classe LOCAL_MATRIX combine donc simplement la spécification abstraite héritée de la classe MATRIX avec les mécanismes de stockage fournis par la classe ARRAY2, l'une des nombreuses classes standard de la librairie Eiffel.

```
class LOCAL_MATRIX inherit
  MATRIX
    redefine nrow, ncolumn end;
  ARRAY2 [DOUBLE]
    rename height as nrow, width as ncolumn end;
creation
  make
end -- class LOCAL_MATRIX
```

La classe LOCAL_MATRIX tient en seulement quelques lignes. On voit là un exemple typique d'utilisation du mécanisme d'héritage multiple : la spécification abstraite héritée de la classe MATRIX est combinée avec les facilités de mise en œuvre offertes par une classe standard de la librairie Eiffel. L'effort de développement se limite donc ici à assurer la correspondance entre les noms des méthodes et attributs hérités des deux classes ancêtres (les attributs `height` et `width` de la classe ARRAY2 sont ici mis en correspondance avec les attributs `nrow` et `ncolumn` de la classe MATRIX).

À la différence de la classe abstraite MATRIX, la classe LOCAL_MATRIX est une classe concrète (ou effective), ce qui signifie qu'elle peut être instanciée. On peut donc créer des matrices locales dans un programme d'application, et les manipuler

en invoquant les opérateurs hérités de la classe MATRIX. Dans l'exemple suivant, on crée ainsi deux matrices locales `A` et `B` de taille 10×10, après quoi on initialise les matrices `A` et `B` de manière aléatoire (l'opérateur `random` est l'un des opérateurs définis dans la classe MATRIX) et on ajoute la matrice `B` à la matrice `A`.

```
local
  A, B: MATRIX;
do
  !LOCAL_MATRIX!A.make (10, 10);
  !LOCAL_MATRIX!B.make (10, 10);
  A.random; B.random;
  A.add (B);
end;
```

Les *références* `A` et `B` sont déclarées ici comme ayant pour *type de base* le type MATRIX. Ce n'est que lors de la création effective des matrices que l'on spécifie le *type dynamique* de `A` et de `B`, qui se trouvent alors associées à des objets de type LOCAL_MATRIX.

2.4 Mise en œuvre des matrices distribuées

2.4.1 Schéma de distribution

Les matrices distribuées de la bibliothèque PALADIN sont partitionnées en blocs, lesquels sont ensuite affectés aux nœuds de la machine parallèle cible, conformément aux directives de distribution fournies par l'utilisateur. Le paramétrage de la distribution des matrices est inspiré de la syntaxe utilisée dans la directive `DISTRIBUTE` du langage HPF. L'utilisateur décrit un schéma de distribution en spécifiant la taille du domaine d'indices considéré, la taille des blocs de partitionnement, et en indiquant dans quel sens ces blocs doivent être pris en compte lors de leur affectation aux nœuds. La figure 2 illustre la création d'une matrice distribuée.

La gestion de la distribution implique un certain nombre de calculs élémentaires mais répétitifs, tels que ceux visant à déterminer l'identité du nœud propriétaire d'un élément (i, j) donné et son adresse locale sur ce nœud. Les méthodes permettant d'effectuer ces calculs ont été encapsulées dans une classe baptisée DISTRIBUTION_2D. Lors de sa création, chaque matrice distribuée va être associée à une instance de la classe DISTRIBUTION_2D, qui lui servira de *template*. Une matrice pourra être redistribuée en changeant de *template* dynamiquement, et plusieurs matrices pourront partager un même schéma de distribution en référençant le même *template* (de telles matrices seront alors dites *matrices alignées*).

```
item (i, j: INTEGER): DOUBLE is
  do
    if dist.item_is_local(i, j) then
      Result := local_item (i, j);
      broadcast (Result);
    else
      Result := receive_from (dist.owner_of_item (i, j));
    end; -- if
  end;
```

Dans ces deux extraits de code l'attribut `dist` référence le *template* auquel est associée la matrice courante. Les primitives de communication invoquées dans le code de la fonction `item` font partie des services offerts par la classe POM (voir figure 1), qui assure l'interface entre le langage Eiffel et une librairie de communication développée dans l'équipe PAMPA. Des détails sur cette librairie sont fournis au paragraphe 2.6

À la différence de `put` et `item`, les deux méthodes `local_put` et `local_item` constituent des accesseurs locaux, dont la mise en œuvre dépend étroitement du format interne choisi pour représenter sur chaque nœud les blocs dont il est propriétaire. Ces méthodes sont donc définies dans les classes descendantes de DIST_MATRIX, qui encapsulent les détails relatifs à la représentation interne des matrices distribuées.

2.4.3 Représentation interne des matrices distribuées

Nous avons intégré dans la bibliothèque PALADIN plusieurs classes décrivant des formats de représentation interne alternatifs pour les matrices distribuées.

La classe DBLOCK_MATRIX met en œuvre une matrice distribuée par blocs sous la forme d'une table bi-dimensionnelle de matrices blocs (instances de la classe ARRAY2[LOCAL_MATRIX]). Chaque entrée de cette table correspond à un bloc de partitionnement, stocké en mémoire sous la forme d'une matrice locale (voir figure 3). Une entrée vide dans la table indique que le nœud local n'est pas propriétaire du bloc matrice correspondant. Dans cette classe les accesseurs locaux `local_put` et `local_item` sont définis en tenant compte de l'indirection due à la table des blocs.

Nous avons également inclus dans la bibliothèque des classes décrivant des mises en œuvre spécifiques pour les matrices distribuées par lignes et par colonnes (classes DROW_MATRIX et DCOL_MATRIX dans la figure 1). D'autres mises en œuvre peuvent aisément être ajoutées à la bibliothèque PALADIN. Il suffit d'intégrer dans la

type HPF [4, 1]. Il s'agit, pour l'essentiel, de réduire les domaines d'itération afin que chaque nœud ne s'intéresse qu'aux données dont il est propriétaire, et de limiter le coût des échanges de données entre nœuds (par exemple en vectorisant les communications). Toutefois, alors que les compilateurs-parallèliseurs doivent procéder automatiquement à la parallélisation des algorithmes, celle-ci demeure dans notre approche complètement manuelle (bien qu'on puisse encapsuler dans des classes Eiffel des schémas algorithmiques parallèles, comme décrit dans [9]). La parallélisation de la bibliothèque doit impérativement demeurer transparente pour le programmeur d'application, ce qui ne pose pas les mêmes problèmes selon que les opérateurs considérés admettent ou non des paramètres.

2.5.2 Parallélisation des opérateurs sans paramètres

Considérons par exemple la méthode `checksum`, telle qu'elle a été définie dans la classe MATRIX. L'algorithme associé à cette méthode dans la classe MATRIX est purement séquentiel et ne tient donc absolument pas compte de la distribution possible de la matrice dont on désire connaître la somme de contrôle. Il est possible de redéfinir la méthode `checksum` dans la classe DBLOCK_MATRIX, en la dotant d'un algorithme de calcul mieux adapté aux caractéristiques des matrices distribuées par blocs. Le calcul de la somme de contrôle se ramène alors à une opération classique de réduction : chaque nœud calcule d'abord localement une somme de contrôle partielle pour les blocs matrices dont il est propriétaire, puis diffuse le résultat à l'intention des autres nœuds. Le calcul se termine lorsque toutes les sommes locales ont été reçues et additionnées sur tous les nœuds.

Dès lors que l'on a redéfini dans la classe DBLOCK_MATRIX la méthode `checksum` en la dotant d'un algorithme optimisé (*i.e.* tenant compte de la distribution), le mécanisme de la liaison dynamique propre au langage Eiffel garantit que cet algorithme sera automatiquement sélectionné à l'exécution chaque fois que l'on invoquera la méthode `checksum` sur un objet de type dynamique DBLOCK_MATRIX. Considérons le petit programme SPMD suivant :

```
local
  A, B: MATRIX;
  v, w: DOUBLE;
do
  !LOCAL_MATRIX!A.make (...);
  !DBLOCK_MATRIX!B.make (...);
  ....
  v := A.checksum;        -- The default algorithm is used
```

```
    w := B.checksum;        -- The parallel algorithm is used
end;
```

En supposant que la méthode `checksum` n'a pas été redéfinie dans la classe LOCAL_MATRIX, c'est alors l'algorithme de calcul par défaut (celui de la classe MATRIX) qui va être exécuté pour calculer la somme de contrôle de la matrice `A`. Par contre, c'est l'algorithme de calcul parallèle optimisé encapsulé dans la classe DBLOCK_MATRIX qui va calculer la somme de contrôle de la matrice `B`. Le mécanisme de la liaison dynamique assure donc la transparence des parallélisations du point de vue du programmeur d'application.

2.5.3 Parallélisation des opérateurs avec paramètres

Le mécanisme de la liaison dynamique mis en œuvre dans le langage Eiffel est parfois qualifié de mécanisme de *single dispatching*, car il s'appuie sur le type dynamique du seul objet courant, c'est à dire l'objet référencé en partie gauche dans une expression pointée telle que `A.checksum` ou `A.add(B)`. Ce mécanisme n'est pas suffisant dans le cas des opérateurs impliquant plusieurs opérandes, car il est indispensable que les types des objets passés en paramètres soient pris en compte lors de la sélection. Considérons le cas de l'opérateur d'addition `add`, qui prend en paramètre une matrice et l'ajoute à la matrice courante. La mise en œuvre de cette méthode dans la classe MATRIX est telle que l'on peut additionner deux matrices quelconques `A` et `B` en introduisant dans un programme SPMD l'expression `A.add(B)`. Cependant cette mise en œuvre est purement séquentielle et, par conséquent, particulièrement inefficace dès lors que l'une au moins des deux matrices A et B se trouve être une matrice distribuée. On peut aisément insérer dans la classe DBLOCK_MATRIX un nouvel opérateur, que nous baptisons `add_dblock`, capable d'additionner efficacement deux matrices distribuées par blocs, pourvu qu'elles soient partitionnées en blocs de même taille. L'algorithme de `add_dblock` peut être est exprimé en termes d'opérations portant sur des matrices blocs. On restreint les domaines d'itération de telle manière que chaque nœud ne procède qu'à des calculs locaux. Les communications, lorsqu'elles ne peuvent être évitées, se trouvent vectorisées tout naturellement du fait qu'elles portent sur des matrices blocs et non plus sur de simples éléments.

Il n'est guère souhaitable que l'on impose à un programmeur d'application de désigner explicitement l'opérateur le plus approprié à chaque fois qu'il désire effectuer la somme de deux matrices. Cette approche ne satisferait d'ailleurs pas notre objectif de transparence maximale. L'idéal serait que la sélection de l'algorithme approprié

soit réalisée automatiquement à l'exécution en fonction de toutes les opérandes. Un tel mécanisme de *multiple dispatching* existe dans certains langages à objets (*e.g.* CLOS [5] et Cecil [2]), mais pas dans le langage Eiffel car il se combine mal avec le mécanisme de l'encapsulation. Nous pouvons cependant l'émuler en effectuant des tests explicites sur les types dynamiques des objets passés en paramètres. Ainsi, dans la classe DBLOCK_MATRIX, nous redéfinissons l'opérateur d'addition de manière à tester le type dynamique de la matrice passée en paramètre.

```
class DBLOCK_MATRIX inherit
  DIST_MATRIX
    rename
      add as add_default        -- Keep the default sequential operator
    end
  ...
feature -- Optimized operators
  add (B: MATRIX) is
    local
      tmp_B: DBLOCK_MATRIX;
    do
      tmp_B?= B;   -- tmp_B:= B if B has type 'DBLOCK_MATRIX'
                   -- tmp_B:= Void otherwise
      if (tmp_B /= Void)
        and then (dist.bfi = tmp_B.dist.bfi)        -- Do matrices have
        and then (dist.bfj = tmp_B.dist.bfj) then -- same block size?
          Current.add_dblock (tmp_B);
      else
        Current.add_default (B);
      end; -- if
    end;
  ...
end -- class DBLOCK_MATRIX
```

Si le test révèle que le type dynamique de la matrice `B` n'est pas conforme au type DBLOCK_MATRIX, l'algorithme par défaut hérité de la classe MATRIX (opérateur renommé en `add_default` dans la clause d'héritage) est exécuté. Si par contre `B` s'avère être une matrice distribuée par blocs, et qu'en outre les blocs de `B` ont la même taille que ceux de la matrice courante, alors c'est l'algorithme parallèle optimisé de l'opérateur `add_dblock` qui est invoqué. Dans les deux cas, le coût des tests effectués pour sélectionner l'un ou l'autre algorithme demeure totalement

négligeable par rapport à la complexité des algorithmes effectuant la somme des matrices.

2.6 Portabilité et performances

La portabilité et la performance sont souvent présentées comme deux objectifs incompatibles, que nous tentons de réconcilier dans PALADIN.

Le problème de l'interfaçage avec les divers systèmes d'exploitation des machines parallèles et systèmes distribués est résolu grâce à l'interfaçage de PALADIN avec une machine parallèle virtuelle dont les nœuds communiquent par échanges de messages. La POM (*Parallel Observable Machine* [13]) se présente sous la forme d'une librairie de communication aisément portable, qui a d'ores et déjà été implantée sur plusieurs plateformes (machines Intel iPSC/2 et Paragon XP/S, réseau de stations de travail, simulateur sur station de travail). Une classe Eiffel a été développée dans le cadre du projet EPEE, qui fait office d'interface entre la librairie POM (écrite en C) et le monde Eiffel. Ainsi PALADIN peut être aisément mise en œuvre sur n'importe quelle machine cible, pourvu que la librairie POM y ait au préalable été portée.

Le noyau de calcul BLAS [10] (*Basic Linear Algebra Subroutines*), qui constitue une batterie de routines de calcul pour l'algèbre linéaire, est en général mis en œuvre sur les machines parallèles par les constructeurs eux-mêmes et ce, la plupart du temps, directement en assembleur. Ses performances sont souvent sans commune mesure avec celles que l'on peut obtenir à partir d'un code compilé. Il serait dommage de ne pas exploiter de telles performances dans PALADIN, mais il n'est en revanche pas souhaitable de limiter la souplesse, l'extensibilité et le confort d'utilisation de notre bibliothèque sous prétexte que les routines de BLAS ne peuvent manipuler que des vecteurs et matrices représentés sous la forme de tableaux FORTRAN. Les mécanismes mis en œuvre dans PALADIN permettent fort heureusement de faire en sorte que l'interfaçage avec BLAS soit effectué de manière transparente et n'impose aucune restriction sur le développement ultérieur de la bibliothèque. En développant PALADIN, nous avons fait en sorte que la mise en œuvre des matrices locales décrite dans la classe LOCAL_MATRIX soit compatible avec celle des tableaux FORTRAN. Nous avons ensuite redéfini au niveau de la classe LOCAL_MATRIX certains des opérateurs de manière à ce que des routines de BLAS soient invoquées pour effectuer les calculs lorsque le noyau BLAS est disponible sur la machine cible et que le type des opérandes le permet (le langage Eiffel permet d'invoquer des routines externes pré-compilées). Ainsi, chaque fois que l'on doit par exemple calculer le produit de deux matrices locales, et stocker le résultat dans une troisième matrice du même type, c'est la routine `GEMM` du noyau BLAS qui est invoquée pour effec-

tuer les calculs, au lieu de la routine par défaut exprimée en Eiffel dans la classe MATRIX. En interfaçant ainsi PALADIN avec le noyau BLAS, on a pu augmenter sensiblement les performances globales de la bibliothèque, sans pour autant compromettre la transparence pour le programmeur d'application. Le tableau suivant présente les performances de PALADIN, observées en effectuant quelques produits de matrices carrées représentées en double précision et distribuées par blocs sur la machine Paragon XP/S d'Intel fonctionnant avec le système d'exploitation OSF/1.

Taille des matrices	Nb. Nœuds	Mflops
400×400	1	45.71
400×400	4	121.90
800×800	4	159.75
1200×1200	16	540.85
2000×2000	40	1155.23
2800×2800	56	1599.42

3 Conclusion

Les mécanismes de la programmation par objets permettent de construire des bibliothèques de calcul pour machines parallèles, dans lesquelles la parallélisation par distribution des données est combinée avec le mode d'exécution SPMD. Nous avons décrit la bibliothèque de démonstration PALADIN, bâtie selon cette approche et dédiée au calcul d'algèbre linéaire sur architecture parallèle à mémoire distribuée.

Au cours du développement de PALADIN, les mécanismes de l'encapsulation et du masquage d'information nous ont permis de rendre la distribution des données et le parallélisme afférant transparents pour l'utilisateur. Grâce au mécanisme de l'héritage multiple, la bibliothèque demeure extensible et peut aisément intégrer de nouveaux algorithmes ou de nouveaux formats de représentation pour les matrices et vecteurs. PALADIN étant interfacée avec le noyau BLAS (lorsque celui-ci est disponible sur la machine cible) et avec la librairie de communication POM, elle présente des performances très honorables et peut être aisément portée sur de nouvelles machines parallèles.

La gestion de la distribution des données et celle du parallélisme afférant sont, dans notre approche, totalement dynamiques. Cette caractéristique distingue fondamentalement EPEE des compilateurs-paralléliseurs semi-automatiques de type HPF, dans lesquels la distribution et le parallélisme ne peuvent être gérés efficacement que dans un contexte purement statique (*i.e.* à la compilation). Une extension

intéressante de la bibliothèque PALADIN serait de la doter d'un petit interpréteur, grâce auquel des matrices et vecteurs pourraient être créés et manipulés de manière interactive par l'utilisateur. On disposerait alors d'un outil un peu semblable à *Matlab*, mais capable d'effectuer des calculs d'algèbre linéaire sur une machine parallèle tout en présentant à l'utilisateur une interface purement séquentielle.

Une autre perspective d'étude concerne l'utilisation du mécanisme de redistribution disponible dans la bibliothèque PALADIN. La redistribution s'inscrit principalement dans le cadre de l'optimisation globale d'un programme d'application. Il s'agit de permettre à un utilisateur d'exploiter au mieux le polymorphisme des matrices distribuées, en lui proposant de les redistribuer chaque fois que le besoin s'en fait sentir. Cependant, le coût de la redistribution étant loin d'être négligeable, il est indispensable de ne l'utiliser que de manière judicieuse. Il faut donc être en mesure d'évaluer les performances associées à chaque distribution possible et le coût des redistributions correspondantes. La librairie POM permettant la génération automatique de traces d'exécution, on pourra s'appuyer sur les outils de collecte et d'analyse de traces développés dans le cadre du projet TRACE (projet national du PRC C^3).

Références

[1] F. André, O. Chéron, M. Le Fur, Y. Mahéo, and J.-L. Pazat. Programmation des machines à mémoire distribuée par distribution des données : langages et compilateurs. *Techniques et Sciences Informatiques*, Numéro spécial, "Langages à parallélisme de données", 12(5):563–596, octobre 1993.

[2] C. Chambers. Object-Oriented Multi-Methods in Cecil. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, 1992.

[3] J. Dongarra and al. An Object-Oriented Design for High-Performance Linear Algebra on Distributed Memory Architectures. In *Proceedings of the Object-Oriented Numerics Conference (OON-SKI'93)*, 1993.

[4] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Technical Report Version 1.0, Rice University, May 1993.

[5] R. et al. Gabriel. CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM*, 34(9), 1991.

[6] F. Guidec and J.-M. Jézéquel. Design of a Parallel Object-Oriented Linear Algebra Library. In Karsten M. Decker and René M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 359 – 364, Birkhäuser Verlag, Basel, July 1994. ISBN 3-7643-5090-3.

[7] F. Guidec and J.-M. Jézéquel. Numeric Parallel Programming with Sequential Object Oriented Languages. In *Proceedings of the First Annual Object-Oriented Numerics Conference (OONSKI'93)*, pages 55–69, April 1993.

[8] J.-M. Jézéquel. EPEE: an Eiffel Environment to Program Distributed Memory Parallel Computers. *Journal of Object Oriented Programming*, 6(2):48–54, May 1993.

[9] J.-M. Jézéquel. Transparent parallelisation through reuse: between a compiler and a library approach. In O. M. Nierstrasz, editor, *ECOOP'93 proceedings*, pages 384–405, Lecture Notes in Computer Science, Springer Verlag, July 1993.

[10] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran. *ACM Transactions on Math. Software*, 14:308–325, 1989.

[11] G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Les langages à objets*. InterÉditions, 1989.

[12] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[13] Pampa Irisa. Un support d'exécution pour machines parallèles. In *Actes des 6èmes rencontres francophones du parallélisme*, pages 207–212, RenPar'6 (Éd. Luc Bougé), juin 1994.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399